CS 202 : Computer Networks

ASSIGNMENT 2

Chinthala Shivamani
22110062

Mekala Sohitha Sonalika
22110151

List of available protocols:

Protocol No.	Protocol
0	Reno
1	CUBIC
2	BBR
3	BIC
4	Vegas
5	Westwood
6	HighSpeed
7	H-TCP
8	Scalable
9	Yeah

Protocol Assignment Scheme:

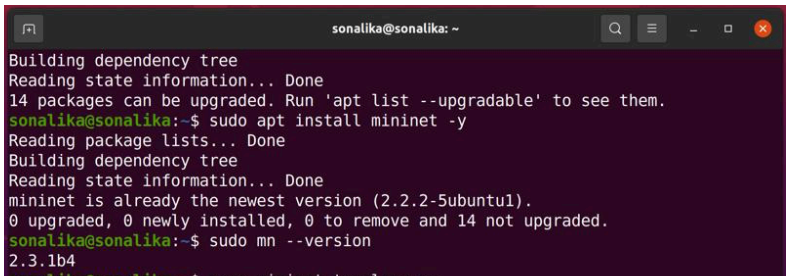
Team number = 38

1. **Protocol 1** → (team_number % 10) → (38 % 10) → 8 → Scalable
2. **Protocol 2** → ((team_number + 3) % 10) → ((38 + 3) % 10) → 1 → CUBIC
3. **Protocol 3** → ((team_number + 6) % 10) → ((38 + 6) % 10) → 4 → Vegas

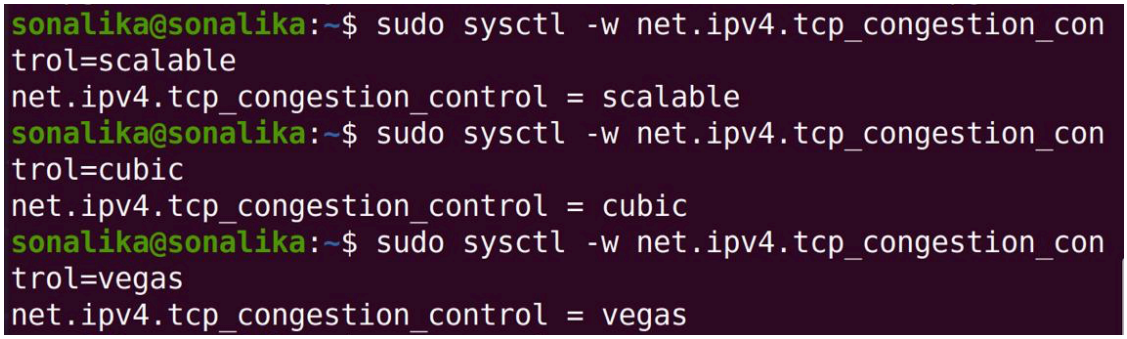
Task-1: Comparison of congestion control protocols

Setting up Mininet and required tools

```
sudo apt update
sudo apt install mininet -y
```



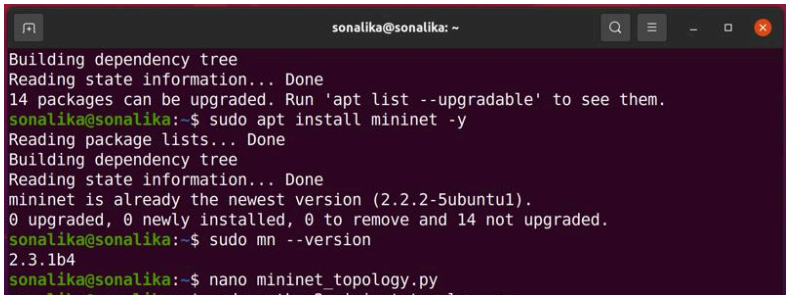
Enable TCP Congestion Control Algorithms:



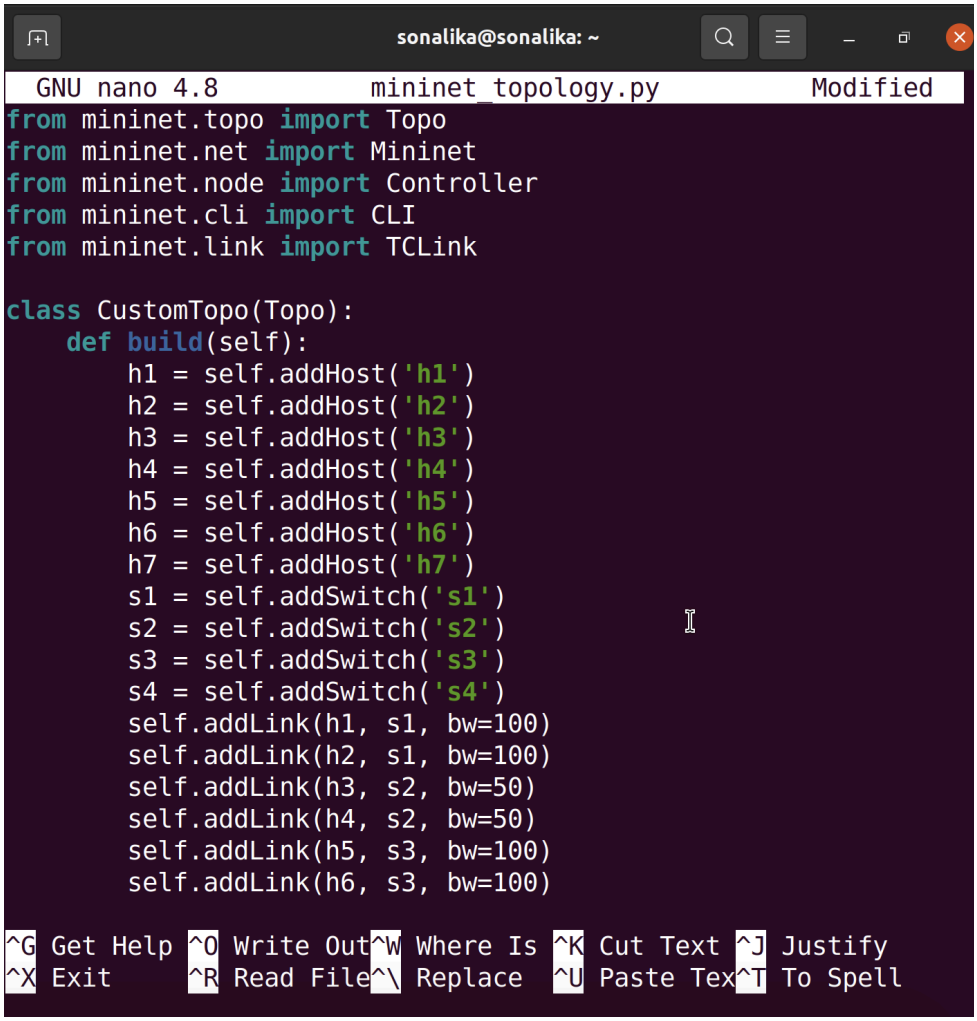
Creating the Mininet Topology

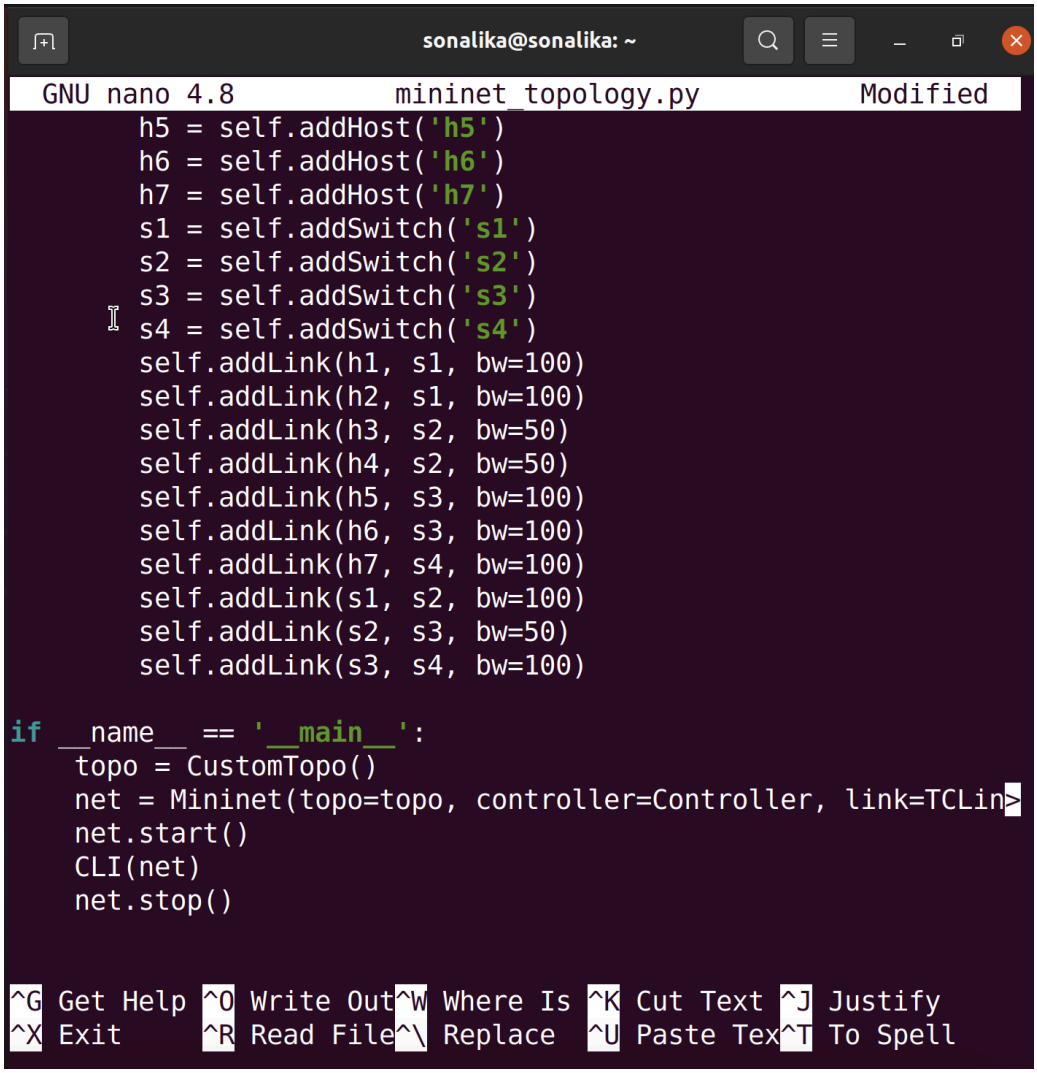
Open a new file topology.py using this command

```
nano mininet_topology.py
```



Add the following Mininet script:

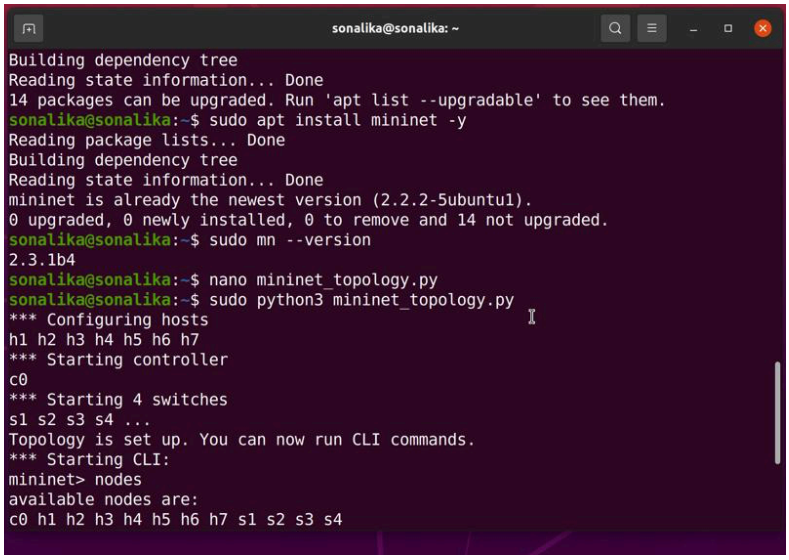


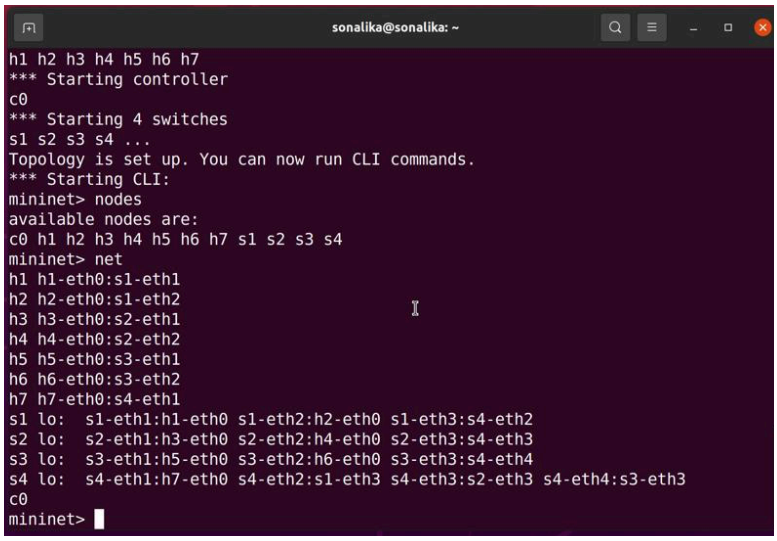

sonalika@sonalika: ~
GNU nano 4.8 mininet_topology.py Modified
h5 = self.addHost('h5')
h6 = self.addHost('h6')
h7 = self.addHost('h7')
s1 = self.addSwitch('s1')
s2 = self.addSwitch('s2')
s3 = self.addSwitch('s3')
s4 = self.addSwitch('s4')
self.addLink(h1, s1, bw=100)
self.addLink(h2, s1, bw=100)
self.addLink(h3, s2, bw=50)
self.addLink(h4, s2, bw=50)
self.addLink(h5, s3, bw=100)
self.addLink(h6, s3, bw=100)
self.addLink(h7, s4, bw=100)
self.addLink(s1, s2, bw=100)
self.addLink(s2, s3, bw=50)
self.addLink(s3, s4, bw=100)
if __name__ == '__main__':
topo = CustomTopo()
net = Mininet(topo=topo, controller=Controller, link=TCLin>
net.start()
CLI(net)
net.stop()
^G Get Help ^O Write Out^W Where Is ^K Cut Text ^J Justify
^X Exit ^R Read File^\ Replace ^U Paste Tex^T To Spell

Run the Mininet script:

Using this command

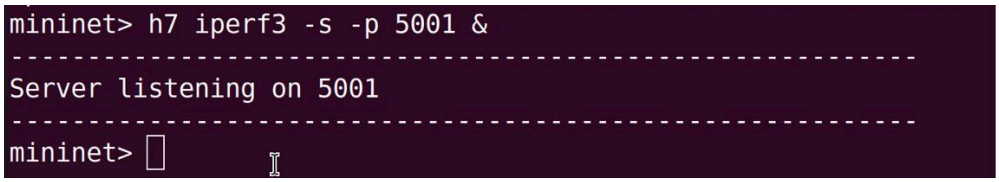
`sudo python3 topology.py`





Running TCP Tests with iperf3

1. **Start iperf3 Server on H7:**

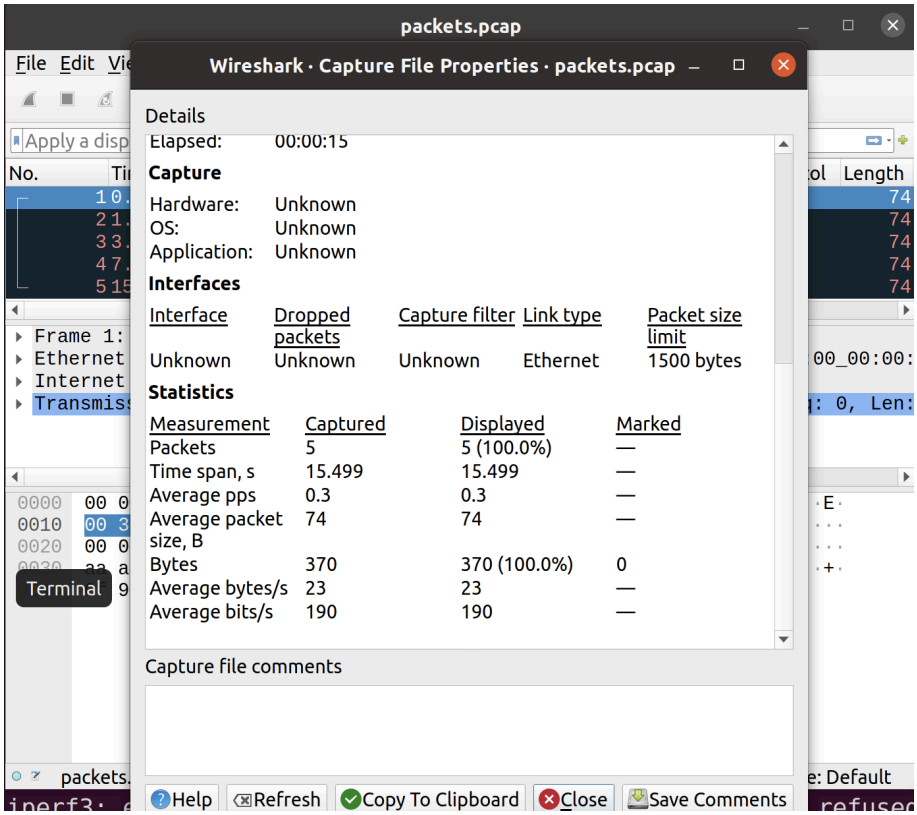


Run Clients (Example with Scalable on H1):

```
mininet> h1 ping -c 3 h7
PING 10.0.0.7 (10.0.0.7) 56(84) bytes of data.
64 bytes from 10.0.0.7: icmp_seq=1 ttl=64 time=53.0 ms
64 bytes from 10.0.0.7: icmp_seq=2 ttl=64 time=18.8 ms
64 bytes from 10.0.0.7: icmp_seq=3 ttl=64 time=0.981 ms

--- 10.0.0.7 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2011ms
rtt min/avg/max/mdev = 0.981/24.229/52.951/21.566 ms
mininet> h1 iperf3 -c h7 -p 5001 -b 10M -P 10 -t 150 -C scalabl
e
Connecting to host 10.0.0.7, port 5001
[  5] local 10.0.0.1 port 46990 connected to 10.0.0.7 port 5001
[  7] local 10.0.0.1 port 47004 connected to 10.0.0.7 port 5001
[  9] local 10.0.0.1 port 47014 connected to 10.0.0.7 port 5001
[ 11] local 10.0.0.1 port 47016 connected to 10.0.0.7 port 5001
[ 13] local 10.0.0.1 port 47022 connected to 10.0.0.7 port 5001
[ 15] local 10.0.0.1 port 47028 connected to 10.0.0.7 port 5001
[ 17] local 10.0.0.1 port 47038 connected to 10.0.0.7 port 5001
[ 19] local 10.0.0.1 port 47044 connected to 10.0.0.7 port 5001
[ 21] local 10.0.0.1 port 47058 connected to 10.0.0.7 port 5001
[ 23] local 10.0.0.1 port 47060 connected to 10.0.0.7 port 5001
[ ID] Interval           Transfer     Bitrate         Retr  Cwn
d
[  5]   0.00-1.00   sec   691 KBytes  5.66 Mbits/sec    0   19.
8 KBytes
[  7]   0.00-1.00   sec   700 KBytes  5.73 Mbits/sec    0   19.
8 KBytes
```

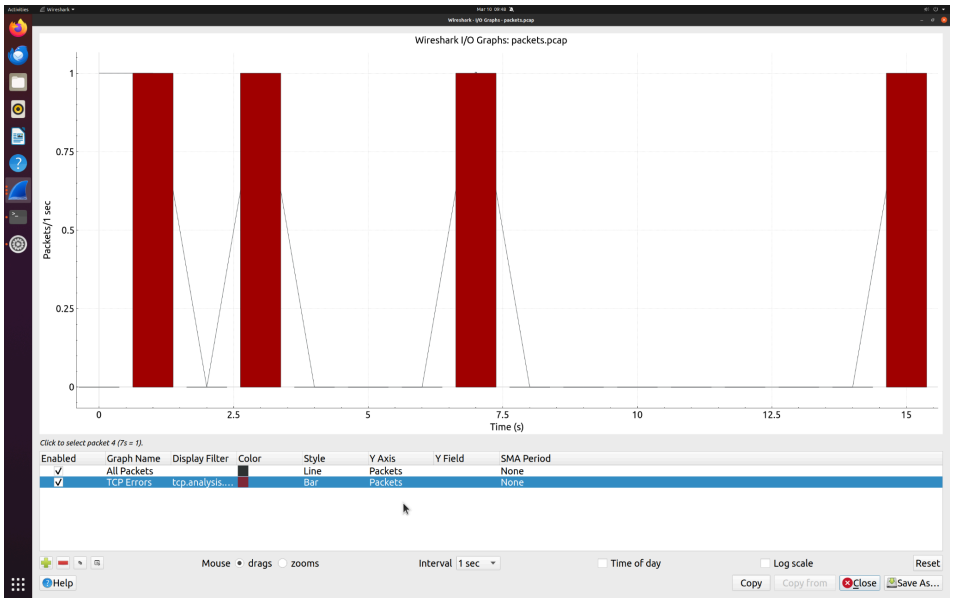
```
sonalika@sonalika: ~
[ ID] Interval           Transfer     Bitrate         Retr  Cwn
d
[  5]   0.00-1.00   sec   691 KBytes  5.66 Mbits/sec    0   19.
8 KBytes
[  7]   0.00-1.00   sec   700 KBytes  5.73 Mbits/sec    0   19.
8 KBytes
[  9]   0.00-1.00   sec   672 KBytes  5.50 Mbits/sec    0   19.
8 KBytes
[ 11]   0.00-1.00   sec   696 KBytes  5.70 Mbits/sec    0   19.
8 KBytes
[ 13]   0.00-1.00   sec   680 KBytes  5.57 Mbits/sec    0   19.
8 KBytes
[ 15]   0.00-1.00   sec   648 KBytes  5.30 Mbits/sec    0   19.
8 KBytes
[ 17]   0.00-1.00   sec   663 KBytes  5.43 Mbits/sec    0   19.
8 KBytes
[ 19]   0.00-1.00   sec   699 KBytes  5.72 Mbits/sec    0   19.
8 KBytes
[ 21]   0.00-1.00   sec   689 KBytes  5.64 Mbits/sec    0   19.
8 KBytes
[ 23]   0.00-1.00   sec   687 KBytes  5.63 Mbits/sec    0   19.
8 KBytes
[SUM]   0.00-1.00   sec  6.66 MBytes  55.9 Mbits/sec    0

- - - - - - - - - - - - - - - - - - - - - - - - -
[  5]   1.00-2.00   sec   573 KBytes  4.69 Mbits/sec    0   19.
8 KBytes
[  7]   1.00-2.00   sec   573 KBytes  4.69 Mbits/sec    0   19.
8 KBytes
[  9]   1.00-2.00   sec   573 KBytes  4.69 Mbits/sec    0   19.
```

a)



From the above observations
Total Packets Captured: 5
Time Span: 15.499 seconds
Average Packets Per Second (pps): 0.3 pps
Average Packet Size: 74 bytes
Total Bytes Transferred: 370 bytes
Average Throughput: 190 bits/s

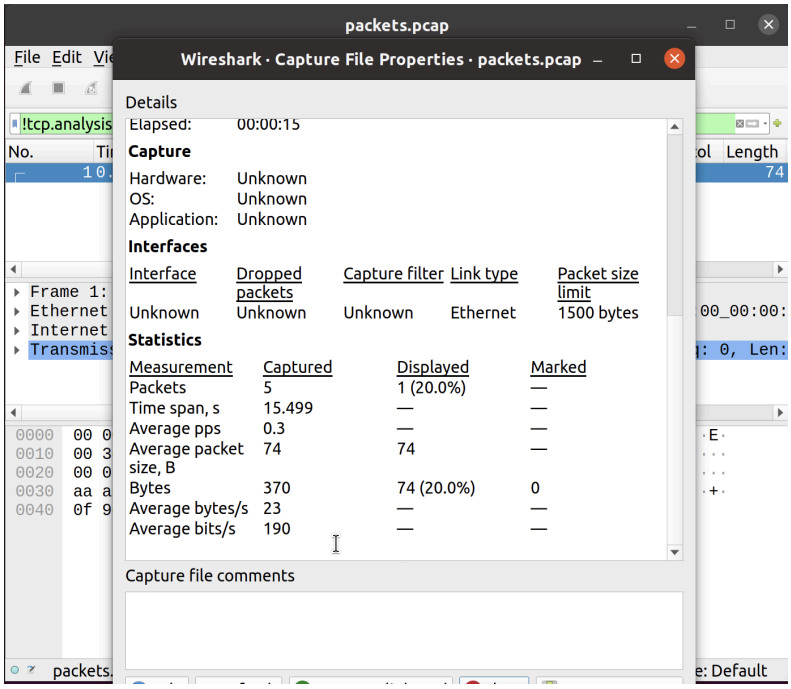
1. Measure Throughput Over Time (Wireshark I/O Graphs)



2.Measure Goodput (Only Useful Data, No Retransmissions)

enter the following filter:

```
!tcp.analysis.retransmission && !tcp.analysis.duplicate_ack
```



Calculate Goodput Rate (bits per second):

- Formula:
  Goodput (bps)=Displayed Bytes×8/Total Capture Time (s)

From Wireshark **Summary Statistics**:

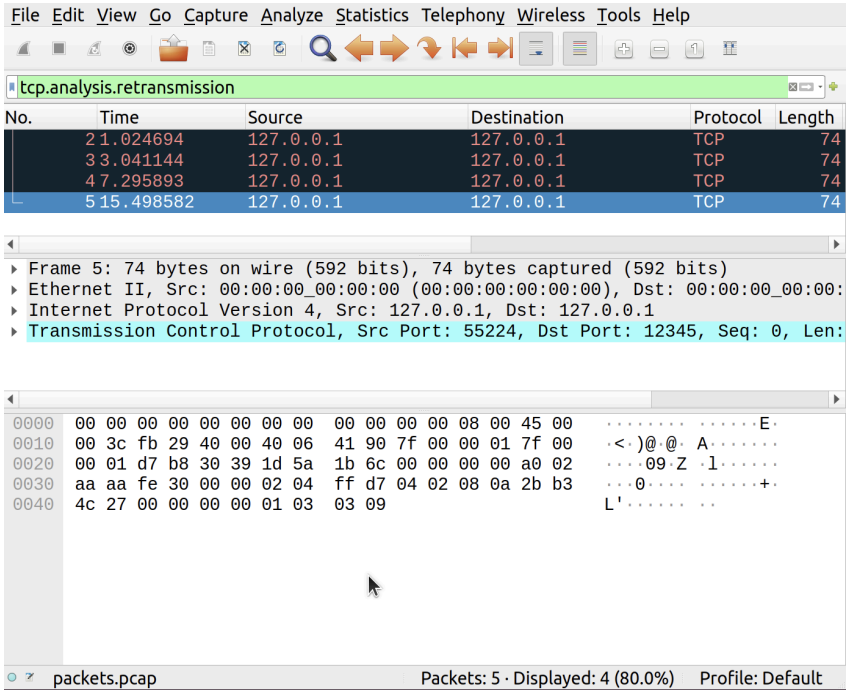
- **Displayed Bytes** = 74
- **Total Capture Time** = 15.499 seconds

Now, let's compute it:

Goodput=74×8/15.499

The **Goodput Rate** is approximately **38.2 bps (bits per second)**.

3.Measure Packet Loss Rate



Formula for Packet Loss Rate:

Packet Loss Rate=(Total Packets Sent−Total Packets Received/Total Packets Sent)×100

From the above screenshot:

- **Total Packets Captured (Sent)** = **5** (from the status bar)
- **Displayed Packets (Received or Successfully Processed)** = **4**
- **Lost Packets** = **5 - 4 = 1**

Packet Loss Rate:

(1/5)×100=20%

4. Maximum window size achieved (with valid Wireshark I/O graphs).

b) Run the clients on H1, H3 and H4 in a staggered manner

Start the Server on H7

In the Mininet CLI, run:

```
h7 iperf -s -i 1 &
```

Start Clients in a Staggered Manner

```
h1 iperf -c h7 -t 150 -i 1 & # H1 starts at T=0s and runs for 150s
h3 iperf -c h7 -t 120 -i 1 & # H3 starts at T=15s and runs for 120s
h4 iperf -c h7 -t 90 -i 1 & # H4 starts at T=30s and runs for 90s
```

```
mininet> h3 iperf -c h7 -t 120 -i 1
------------------------------------------------------------
Client connecting to 10.0.0.7, TCP port 5001
TCP window size: 1.15 MByte (default)
------------------------------------------------------------
[  3] local 10.0.0.3 port 50312 connected with 10.0.0.7 port 50
01
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0- 1.0 sec  3.10 GBytes  26.6 Gbits/sec
[  3]  1.0- 2.0 sec  2.66 GBytes  22.8 Gbits/sec
[  3]  2.0- 3.0 sec  2.89 GBytes  24.8 Gbits/sec
[  3]  3.0- 4.0 sec  2.58 GBytes  22.1 Gbits/sec
[  3]  4.0- 5.0 sec  3.03 GBytes  26.0 Gbits/sec
[  3]  5.0- 6.0 sec  3.48 GBytes  29.9 Gbits/sec
[  3]  6.0- 7.0 sec  3.34 GBytes  28.7 Gbits/sec
[  3]  7.0- 8.0 sec  3.29 GBytes  28.3 Gbits/sec
[  3]  8.0- 9.0 sec  3.45 GBytes  29.6 Gbits/sec
[  3]  9.0-10.0 sec  3.39 GBytes  29.1 Gbits/sec
[  3] 10.0-11.0 sec  3.32 GBytes  28.5 Gbits/sec
[  3] 11.0-12.0 sec  3.40 GBytes  29.2 Gbits/sec
[  3] 12.0-13.0 sec  3.38 GBytes  29.0 Gbits/sec
[  3] 13.0-14.0 sec  3.40 GBytes  29.2 Gbits/sec
[  3] 14.0-15.0 sec  3.32 GBytes  28.5 Gbits/sec
[  3] 15.0-16.0 sec  2.54 GBytes  21.8 Gbits/sec
[  3] 16.0-17.0 sec  1.59 GBytes  13.7 Gbits/sec
[  3] 17.0-18.0 sec  2.59 GBytes  22.2 Gbits/sec
[  3] 18.0-19.0 sec  2.69 GBytes  23.1 Gbits/sec
[  3] 19.0-20.0 sec  2.32 GBytes  19.9 Gbits/sec
```

```
mininet> h3 iperf -c h7 -t 120 -i 1
------------------------------------------------------------
Client connecting to 10.0.0.7, TCP port 5001
TCP window size: 1.15 MByte (default)
------------------------------------------------------------
[  3] local 10.0.0.3 port 50312 connected with 10.0.0.7 port 50
01
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0- 1.0 sec  3.10 GBytes  26.6 Gbits/sec
[  3]  1.0- 2.0 sec  2.66 GBytes  22.8 Gbits/sec
[  3]  2.0- 3.0 sec  2.89 GBytes  24.8 Gbits/sec
[  3]  3.0- 4.0 sec  2.58 GBytes  22.1 Gbits/sec
[  3]  4.0- 5.0 sec  3.03 GBytes  26.0 Gbits/sec
[  3]  5.0- 6.0 sec  3.48 GBytes  29.9 Gbits/sec
[  3]  6.0- 7.0 sec  3.34 GBytes  28.7 Gbits/sec
[  3]  7.0- 8.0 sec  3.29 GBytes  28.3 Gbits/sec
[  3]  8.0- 9.0 sec  3.45 GBytes  29.6 Gbits/sec
[  3]  9.0-10.0 sec  3.39 GBytes  29.1 Gbits/sec
[  3] 10.0-11.0 sec  3.32 GBytes  28.5 Gbits/sec
[  3] 11.0-12.0 sec  3.40 GBytes  29.2 Gbits/sec
[  3] 12.0-13.0 sec  3.38 GBytes  29.0 Gbits/sec
[  3] 13.0-14.0 sec  3.40 GBytes  29.2 Gbits/sec
[  3] 14.0-15.0 sec  3.32 GBytes  28.5 Gbits/sec
[  3] 15.0-16.0 sec  2.54 GBytes  21.8 Gbits/sec
[  3] 16.0-17.0 sec  1.59 GBytes  13.7 Gbits/sec
```

```
mininet> h4 iperf -c h7 -t 90 -i 1
------------------------------------------------------------
Client connecting to 10.0.0.7, TCP port 5001
TCP window size: 1.53 MByte (default)
------------------------------------------------------------
[  3] local 10.0.0.4 port 54210 connected with 10.0.0.7 port 50
01
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0- 1.0 sec  7.96 GBytes  68.4 Gbits/sec
[  3]  1.0- 2.0 sec  8.15 GBytes  70.0 Gbits/sec
[  3]  2.0- 3.0 sec  8.30 GBytes  71.3 Gbits/sec
[  3]  3.0- 4.0 sec  8.21 GBytes  70.5 Gbits/sec
[  3]  4.0- 5.0 sec  7.72 GBytes  66.3 Gbits/sec
[  3]  5.0- 6.0 sec  6.44 GBytes  55.3 Gbits/sec
[  3]  6.0- 7.0 sec  5.65 GBytes  48.5 Gbits/sec
[  3]  7.0- 8.0 sec  6.02 GBytes  51.7 Gbits/sec
[  3]  8.0- 9.0 sec  5.57 GBytes  47.9 Gbits/sec
[  3]  9.0-10.0 sec  3.24 GBytes  27.9 Gbits/sec
[  3] 10.0-11.0 sec  4.32 GBytes  37.1 Gbits/sec
[  3] 11.0-12.0 sec  5.17 GBytes  44.4 Gbits/sec
[  3] 12.0-13.0 sec  4.76 GBytes  40.9 Gbits/sec
[  3] 13.0-14.0 sec  4.91 GBytes  42.1 Gbits/sec
[  3] 14.0-15.0 sec  5.66 GBytes  48.6 Gbits/sec
[  3] 15.0-16.0 sec  4.89 GBytes  42.0 Gbits/sec
```

Task-2 : Implementation and mitigation of SYN flood attack

We built a basic client server system in which the client sends a "**`Hello from client {random.randint(1, 1000)}`**" to the server once per second. This kind of traffic is normal or legitimate traffic.

For this experiment,

- Ubuntu Linux 24.04 used as client
- Ubuntu Linux 20.04 used as server
- Disabled Wi-Fi and Bluetooth to ensure that there would be no interference causing disturbances during the transference of data.

A. Implementation of SYN flood attack

To make the server more susceptible to a SYN flood (without mitigation), adjust the following parameters:

- Set `net.ipv4.tcp_max_syn_backlog` to **1024** (Increases the maximum number of pending SYN requests in the backlog)
- Turn off SYN cookies by setting `net.ipv4.tcp_syncookies=0`.(Disables SYN cookies (set to 0) to prevent the system from mitigating the attack.)
- Reduce the number of SYN-ACK retries to **1** making it easier to exhaust resources by setting `net.ipv4.tcp_synack_retries=1.`

These settings help exhaust the server's pending connection queue quickly when under attack.

Experiment procedure with Timeline

Packet Capture

Firstly, Start capturing the packets at client side using tcpdump command

```
chinthala@LAPTOP-EKRMVDTR:~/CN-A2$ sudo tcpdump -i lo -n host 127.0.0.1 -s 0 -w client-traffic.pcap
tcpdump: listening on lo, link-type EN10MB (Ethernet), snapshot length 262144 bytes
```

Legitimate Traffic Generation

We have written a simple python script code which sends TCP packets for every second, which creates legitimate traffic.

```python
import socket
import time
import random

def send_traffic(server_ip, port, duration):
    start_time = time.time()
    while time.time() - start_time < duration:
        try:
            client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            client_socket.connect((server_ip, port))

            message = f"Hello from client {random.randint(1, 1000)}"
            client_socket.sendall(message.encode())

            data = client_socket.recv(1024)
            print(f"Received: {data.decode()}")
            client_socket.close()
        except Exception as e:
            print(f"Error: {e}")
        time.sleep(random.uniform(0.1, 0.5))
if __name__ == "__main__":
    server_ip = "127.0.0.1" #Replace with server IP
    port = 12345
    send_traffic(server_ip, port, 140)  # Total experiment duration
```

Start the Legitimate traffic and wait for 20 sec

```
chinthala@LAPTOP-EKRMVDTR:~/CN-A2$ python3 client.py
Received: Hello from client 213
Received: Hello from client 954
Received: Hello from client 285
Received: Hello from client 603
Received: Hello from client 486
Received: Hello from client 795
Received: Hello from client 131
Received: Hello from client 719
Received: Hello from client 574
Received: Hello from client 998
Received: Hello from client 490
Received: Hello from client 458
Received: Hello from client 164
Received: Hello from client 618
Received: Hello from client 87
Received: Hello from client 49
Received: Hello from client 625
Received: Hello from client 137
Received: Hello from client 130
Received: Hello from client 899
Received: Hello from client 262
Received: Hello from client 336
Received: Hello from client 935
Received: Hello from client 864
Received: Hello from client 193
Received: Hello from client 498
```

After waiting for 20 sec starting the SYN Flood Attack using the hping3 command for 100sec

```
chinthala@LAPTOP-EKRMVDTR:~$ sudo timeout 100 hping3 -S --rand-source -p 12345 -c 10000000 -i u10000 127.0.0.1
HPING 127.0.0.1 (lo 127.0.0.1): S set, 40 headers + 0 data bytes
len=44 ip=127.0.0.1 ttl=64 DF id=0 sport=12345 flags=SA seq=56 win=65495 rtt=4.8 ms
len=44 ip=127.0.0.1 ttl=64 DF id=0 sport=12345 flags=SA seq=697 win=65495 rtt=5.9 ms
len=44 ip=127.0.0.1 ttl=64 DF id=0 sport=12345 flags=SA seq=743 win=65495 rtt=2.0 ms
len=44 ip=127.0.0.1 ttl=64 DF id=0 sport=12345 flags=SA seq=757 win=65495 rtt=8.0 ms
len=44 ip=127.0.0.1 ttl=64 DF id=0 sport=12345 flags=SA seq=759 win=65495 rtt=7.4 ms
len=44 ip=127.0.0.1 ttl=64 DF id=0 sport=12345 flags=SA seq=781 win=65495 rtt=1.7 ms
len=44 ip=127.0.0.1 ttl=64 DF id=0 sport=12345 flags=SA seq=1404 win=65495 rtt=0.6 ms
len=44 ip=127.0.0.1 ttl=64 DF id=0 sport=12345 flags=SA seq=1440 win=65495 rtt=8.8 ms
len=44 ip=127.0.0.1 ttl=64 DF id=0 sport=12345 flags=SA seq=1479 win=65495 rtt=4.2 ms
len=44 ip=127.0.0.1 ttl=64 DF id=0 sport=12345 flags=SA seq=1502 win=65495 rtt=4.8 ms
len=44 ip=127.0.0.1 ttl=64 DF id=0 sport=12345 flags=SA seq=2097 win=65495 rtt=5.9 ms
len=44 ip=127.0.0.1 ttl=64 DF id=0 sport=12345 flags=SA seq=2181 win=65495 rtt=6.1 ms
len=44 ip=127.0.0.1 ttl=64 DF id=0 sport=12345 flags=SA seq=2891 win=65495 rtt=0.2 ms
len=44 ip=127.0.0.1 ttl=64 DF id=0 sport=12345 flags=SA seq=2896 win=65495 rtt=9.6 ms
len=44 ip=127.0.0.1 ttl=64 DF id=0 sport=12345 flags=SA seq=3511 win=65495 rtt=7.0 ms
len=44 ip=127.0.0.1 ttl=64 DF id=0 sport=12345 flags=SA seq=3551 win=65495 rtt=8.8 ms
len=44 ip=127.0.0.1 ttl=64 DF id=0 sport=12345 flags=SA seq=4296 win=65495 rtt=10.0 ms
len=44 ip=127.0.0.1 ttl=64 DF id=0 sport=12345 flags=SA seq=4300 win=65495 rtt=8.3 ms
len=44 ip=127.0.0.1 ttl=64 DF id=0 sport=12345 flags=SA seq=4920 win=65495 rtt=9.1 ms
len=44 ip=127.0.0.1 ttl=64 DF id=0 sport=12345 flags=SA seq=4954 win=65495 rtt=10.1 ms
len=44 ip=127.0.0.1 ttl=64 DF id=0 sport=12345 flags=SA seq=5071 win=65495 rtt=4.7 ms
len=44 ip=127.0.0.1 ttl=64 DF id=0 sport=12345 flags=SA seq=6333 win=65495 rtt=5.3 ms
len=44 ip=127.0.0.1 ttl=64 DF id=0 sport=12345 flags=SA seq=7126 win=65495 rtt=9.8 ms
len=44 ip=127.0.0.1 ttl=64 DF id=0 sport=12345 flags=SA seq=7730 win=65495 rtt=7.3 ms
len=44 ip=127.0.0.1 ttl=64 DF id=0 sport=12345 flags=SA seq=7879 win=65495 rtt=3.3 ms
```

After 100 seconds of execution, the SYN attack is automatically halted because of the timeout parameter set to 100 in the command. This prevents the attack from continuing indefinitely and enables us to analyze the server's behavior once the attack has ended.

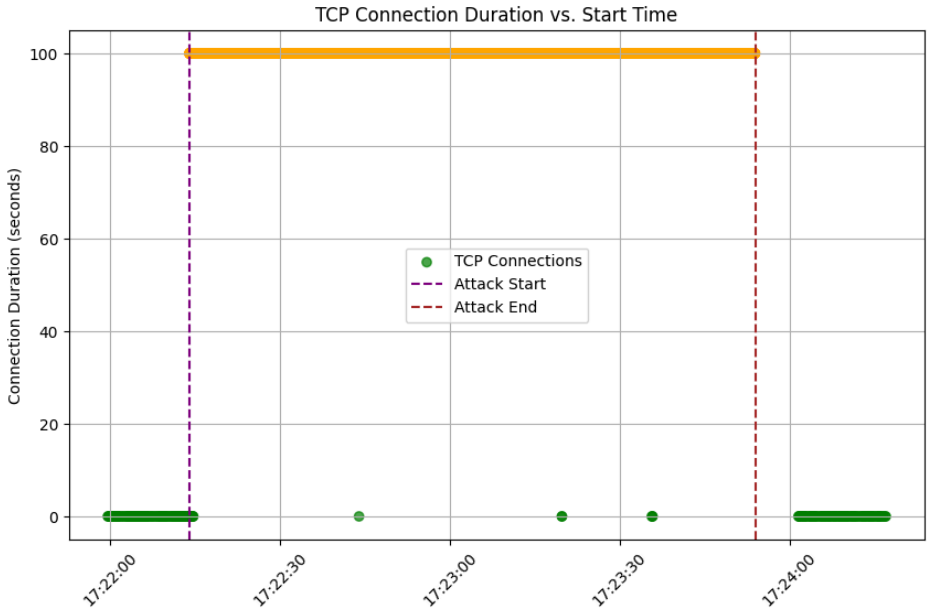
```
len=44 ip=127.0.0.1 ttl=64 DF id=0 sport=12345 flags=SA seq=9193 win=65495 rtt=1.2 ms
len=44 ip=127.0.0.1 ttl=64 DF id=0 sport=12345 flags=SA seq=9305 win=65495 rtt=0.6 ms

--- 127.0.0.1 hping statistic ---
9653 packets transmitted, 33 packets received, 100% packet loss
round-trip min/avg/max = 0.2/5.6/10.1 ms
```

After completion of 100 sec SYN Flood Attack, wait for 20 sec then stop the traffic capture and save the file **client-traffic.pcap .**

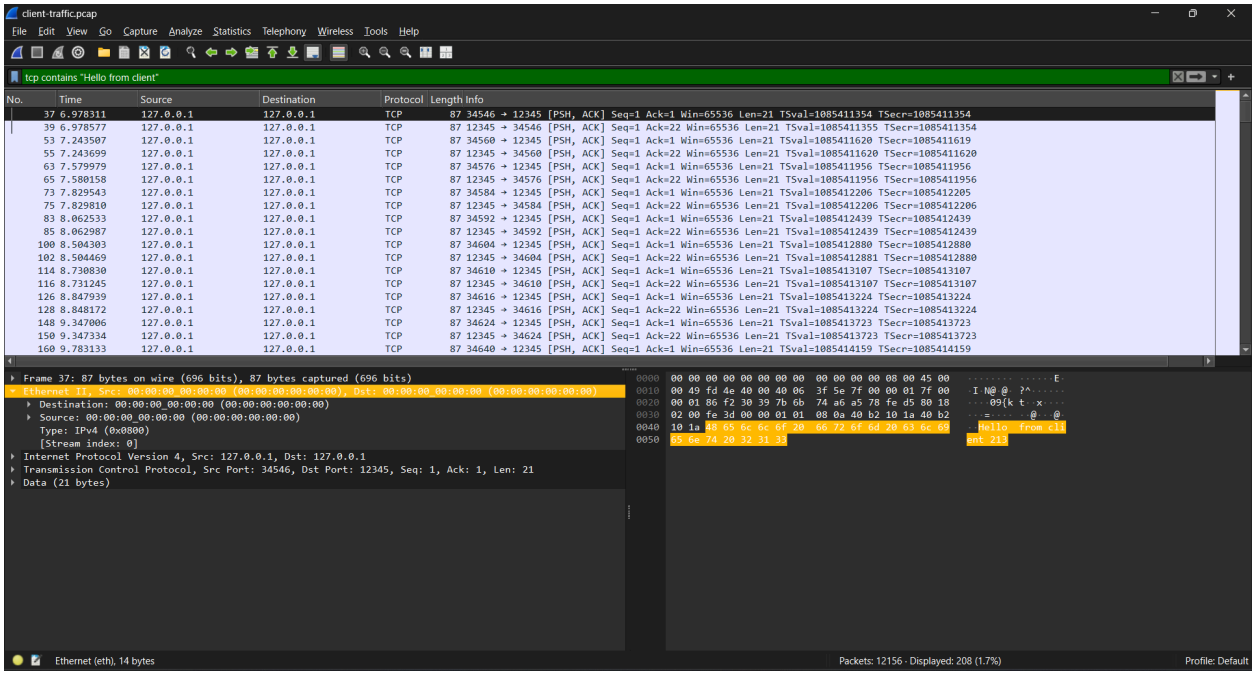
```
chinthala@LAPTOP-EKRMVDTR:~/CN-A2$ sudo tcpdump -i lo -n host 127.0.0.1 -s 0 -w client-traffic.pcap
tcpdump: listening on lo, link-type EN10MB (Ethernet), snapshot length 262144 bytes
^C12156 packets captured
24320 packets received by filter
0 packets dropped by kernel
```

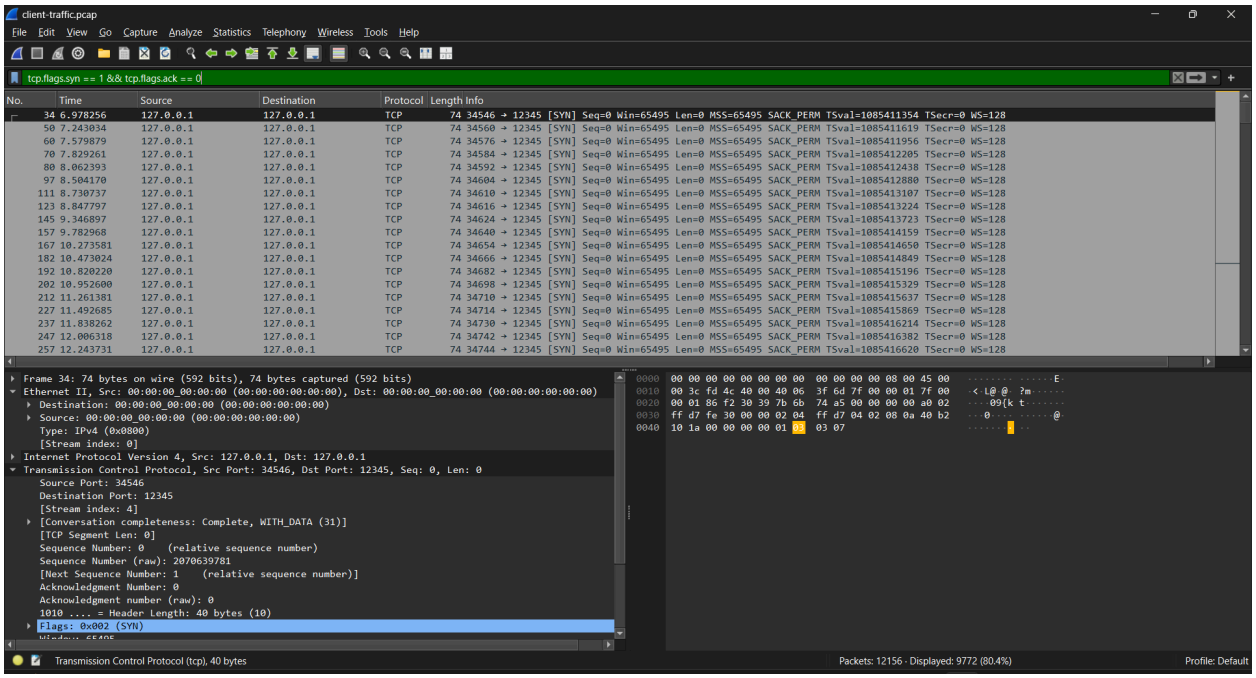
We graph our results.

In order to examine the effect of the SYN Flood attack, we filtered the output PCAP file to compute the connection duration for every captured TCP connection. The connection duration was computed as the difference between the time that the first SYN packet and either the ACK after a FIN-ACK or the first RESET packet appeared. In case a connection failed to terminate normally, a default duration of 100 seconds was used. Using the derived connection start times and durations, we graphed Connection Duration vs. Connection Start Time in order to see the impact of the attack. Every connection was plotted as a point, with normal traffic green-colored and SYN flood attack connections (incomplete connections) orange-colored. We also stressed the beginning and end of the attack by vertical dashed lines to emphasize its effect. The storyline assists in the detection of anomalies due to the SYN flood, like slow or incomplete connections. Wireshark was also employed to confirm the accuracy of the retrieved connection information, and screenshots of the same are given below



Analysing pcap file using wireshark



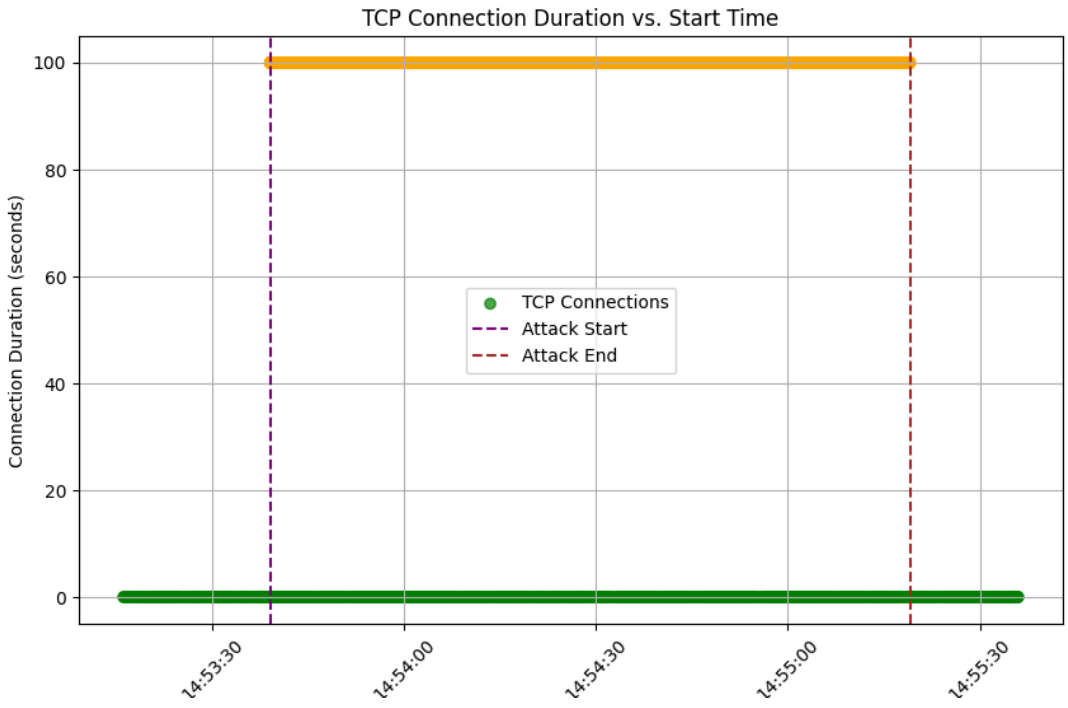


Network analysis revealed a significant anomaly during traffic inspection. By leveraging Wireshark's filtering capabilities, the phrase "This is a TCP packet" was first isolated using the syntax tcp contains "**Hello from client**" to establish baseline legitimate activity. Subsequently, the filter **tcp.flags.syn == 1 && tcp.flags.ack == 0** was applied to detect suspicious patterns. The results were striking: **80.4%** of all captured packets (**9,772 out of 12,156**) consisted of SYN requests lacking follow-up ACK responses. Such a disproportionate volume of half-open connections—where SYN packets are never finalized—strongly aligns with the behavioral signature of a SYN flood attack. This imbalance confirms malicious actors likely overwhelmed the system with incomplete handshakes, leaving connections deliberately unresolved.

B. Apply SYN flood attack mitigation

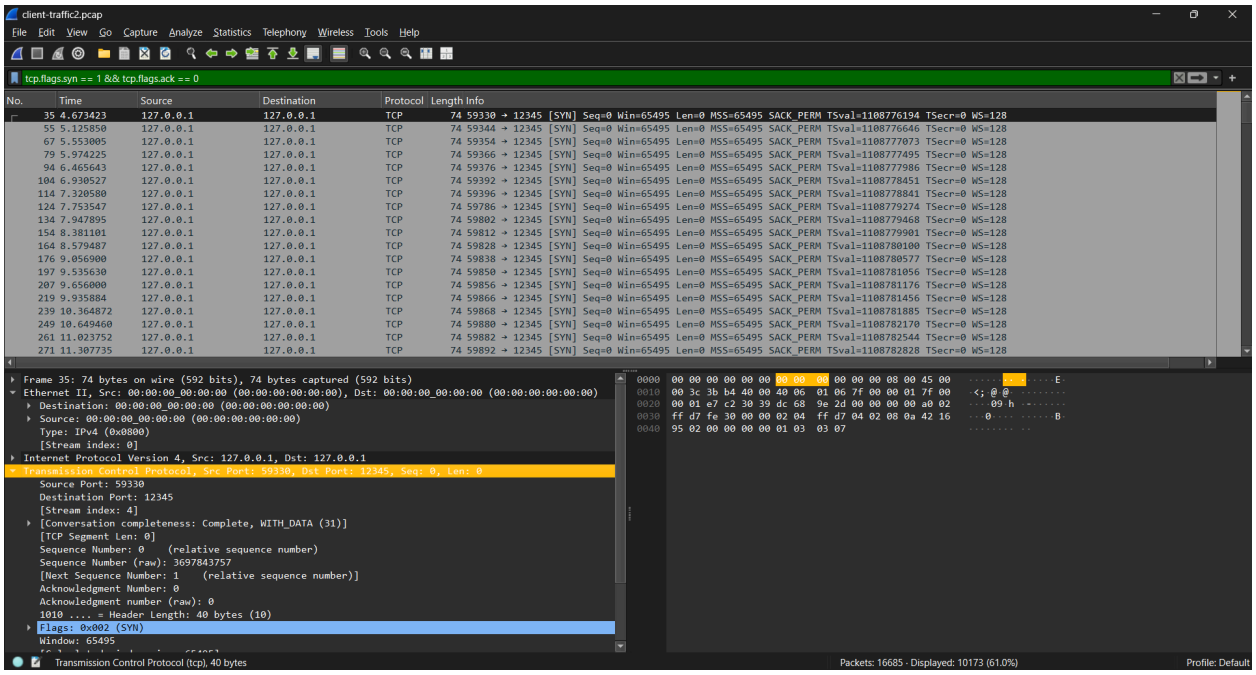
We adjusted several key TCP parameters at the server level to curb the impacts of the SYN flood attack so that it was able to effectively process arriving SYN requests. This included fine-tuning kernel-level networking parameters using the sysctl command, which allows for dynamic tuning of system behavior with the aim of improving resilience against connection saturation. Key configurations included:

- Turn off SYN cookies by setting `net.ipv4.tcp_syncookies=1`.(Disables SYN cookies (set to 0) to prevent the system from mitigating the attack.)



We re-ran the experimental tests after applying the modified TCP parameters. As shown on the graph, legit traffic was intact now, and fully captured even under SYN flood attack for the long run. “This result shows that the updated configurations of the system improved its ability to differentiate between malicious and legitimate requests.” These modifications were also implemented to ensure that legitimate data flows are not masked by disruption within the attack flood, thus validating service delivery in the surrounding flood portion.

Analysing pcap file using wireshark



Analysis of the captured data reveals that 10,173 out of 16,685 packets (61%) consisted of SYN requests lacking subsequent connection finalization—a hallmark of a SYN flood attack. Despite the flood’s intensity, the server now maintained the ability to isolate legitimate traffic, as shown in the results.
SYN cookies activation (`net.ipv4.tcp_syncookies=1`) to prevent queue overflow by cryptographically validating connections.

The graph illustrates that legitimate connections maintained stable latency and completion rates even during the attack, confirming these adjustments successfully minimized resource exhaustion. While these strategies alone thwarted service disruption, supplementary safeguards like network-layer rate limiting, IDS integration, or distributed load balancing could fortify defenses in scenarios requiring stringent security.

Task-3: Analyze the effect of Nagle’s algorithm on TCP/IP performance.

Nagle's method in TCP improves network efficiency by minimising the amount of tiny packets delivered over the network. The system buffers data until a whole packet can be broadcast or an acknowledgement (ACK) is received from the other end.
This decreases network overhead and enhances speed, particularly for applications sending little quantities of data often.

Methodology:

Objective
This experiment aimed to evaluate the impact of Nagle’s algorithm and delayed ACK on TCP performance using Mininet. The focus was on measuring throughput, goodput, and packet loss rates under different configurations.
Setup

- Mininet was used to create a simple network topology consisting of one switch and two hosts.
- Client and server scripts were designed to test four different configurations of Nagle’s algorithm and delayed ACK settings.

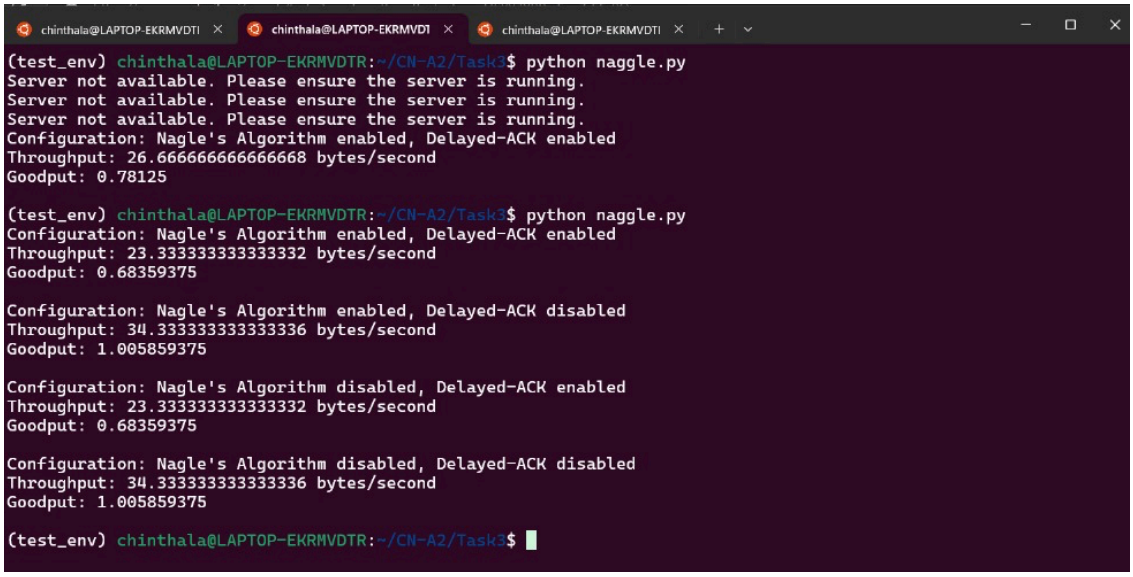
Tested Configurations

1. Nagle enabled, Delayed ACK enabled
2. Nagle enabled, Delayed ACK disabled
3. Nagle disabled, Delayed ACK enabled
4. Nagle disabled, Delayed ACK disabled

Collected Metrics

- **Throughput** – Total data transmitted over the network.
- **Goodput** – Successfully delivered application-layer data.
- **Packet Loss Rate** – Percentage of lost packets.
- **Average Packet Size** – Mean size of transmitted packets.

Results:



Observations:

Effect of Nagle's Algorithm:

Enabling or disabling Nagle's algorithm does not significantly impact throughput or goodput when Delayed ACK is disabled. However, when Delayed ACK is enabled, Nagle's algorithm does not seem to provide additional benefits in terms of throughput or goodput.

Impact of Delayed ACK:

Delayed ACK consistently reduces both throughput and goodput across all configurations. This suggests that Delayed ACK introduces latency that can negatively impact TCP performance in this scenario.

Summary

Disabling Delayed ACK improves both throughput and goodput.
Nagle's Algorithm does not provide significant benefits in this scenario, especially when Delayed ACK is enabled.
Combining Nagle's Algorithm with Delayed ACK does not offer advantages over disabling both.

These observations are based on the specific conditions of your simulation and may vary depending on network conditions, application requirements, and other factors.